2011 Fourth IEEE International Conference on Software Testing, Verification and Validation

A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications

Sara Sprenkle*, Lori Pollock†, Lucy Simko*

* Dept of Computer Science
Washington and Lee University
Lexington, VA USA
sprenkles@wlu.edu, simkol11@mail.wlu.edu

† Dept of Computer & Information Sciences
University of Delaware
Newark, DE USA
pollock@cis.udel.edu

Abstract

While web applications expand in usage and complexity, testing demands are growing without corresponding automated support. One promising approach to automatic test generation is statistical model-based testing, where logged user behavior is used to build a usage-based model of web application navigation, from which abstract test cases are generated. Executable test cases are then created by adding parameter values to the abstract test cases. Several researchers have proposed variations of this approach; however, no one has empirically examined the tradeoffs and implications of the different ways to represent user behavior in a navigation model and the characteristics of the automatically generated test cases from different models.

We report on our exploratory study of automatically generated abstract test cases and the underlying usage-based navigation models constructed from over 3500 user sessions across five publicly deployed web applications. Our results suggest how web testers can easily tune statistical model-based automatic test case generators for web applications toward generating tests closely related to user behavior or toward new navigations without using large additional test resources.

1. Introduction

As web applications are used more pervasively and their size and complexity increase, ensuring their correctness becomes more critical and more challenging. Practical static analysis of web applications is inhibited by the dynamic features of the languages, dynamically generated web pages, complex application state dependencies and concurrent user interactions. Thus, effective runtime testing is crucial.

One particular property of web applications creates a unique testing challenge: application navigation. Unlike traditional applications, users can circumvent the application's desired navigation constraints by utilizing the browser's features, such as the back button, location bar, bookmarks, or multiple windows. Tonella and Ricca [19] found that 47% of user sessions included an "infeasible" navigation, which is a navigation that does not follow any edge in their extracted model of the web application, presumably caused by browser-based navigation. For example, a user may have bookmarked a page with restricted access after they logged in. If the web application does not return an access denial error or request for a login when the user accesses the bookmark after the session times out, the web application has a fault. Further examples of navigation errors are discussed by Halle et al [11]. Proper enforcement of navigation constraints is important and should be tested in addition to legal navigation paths through the application.

Some researchers proposed building a *navigation model* of the application [12, 19, 17, 20] for use in testing and application understanding, among other tasks. Strategies for constructing navigation models of web applications are to generate them using augmented web crawlers [19, 20] or from the web application's usage logs [12, 19, 17]. In testing, the navigation model is used to generate *abstract test cases*, which represent URL navigation sequences. The abstract test cases are converted into *executable test cases* by adding a set of parameter values to be sent as name-value pairs in each request. Since testing all possible navigations is not practical, usage information, which is cheap to obtain, is important to learn what navigation sequences users access. A study of a small web application indicates that usage-based, statistical model-based testing is a promising approach for (1) creating tests for the most (and least) likely user paths through the application, (2) creating tests that include browser-based inputs, (3) generating fewer tests than capturing user logs and replaying them directly, and (4) combining several users' paths through an application [17]. Another case study demonstrates using usage-based, statis-

978-0-7695-4342-0/11 $26.00 © 2011 IEEE
DOI 10.1109/ICST.2011.34

tical model-based testing for reliability estimations [19].

While researchers of statistical model-based testing share similar overall testing and analysis goals, they have differed in their definition of the navigation model. Moreover, no one has empirically examined the tradeoffs and implications of the different ways to represent user behavior in a navigation model nor has there been a study of the characteristics of the automatically generated test cases from different models. Without this information, it is not clear how best to achieve effective testing through the general statistical model-based approach to testing web applications for different testing goals. Furthermore, the navigation model and generated abstract test cases form the basis for generating executable test cases.

In this paper, we investigate how variations of a navigation model can affect the model's representation of user behavior, model size, and the generated abstract test cases. We performed an empirical study of automatically generated abstract test cases and the underlying usage-based navigation models constructed from over 3500 user sessions across five publicly deployed web applications, spanning different technologies and domains.

To the best of our knowledge, no previous work investigated the variations in statistical model-based testing of web applications. Thus, the main contributions of this paper are

- Results from an empirical study of navigation model variations that indicate that users' navigation behavior is not easily approximated without collecting usage information; however, relatively small user logs are needed to build a usage-based navigation model that enables generating test cases highly representative of user behavior.
- Results from studying automatically generated model-based test cases that suggest how web testers can tune the generators toward generating tests closely related to user behavior or toward new navigations without using large additional test resources.

2. Statistical Model-based Testing

2.1. Test Case Generation Process

Figure 1 shows an overview of the test-case generation process that we are focusing on. The test-case generation process begins with logs of user interactions with a web application, then builds navigation and data models, which generate test cases for the web application. Broadly defined, a web application is a set of web pages and components that form a system in which user input (navigation and data input) affects the system's state. Users interact with a web application using a browser, making *requests*

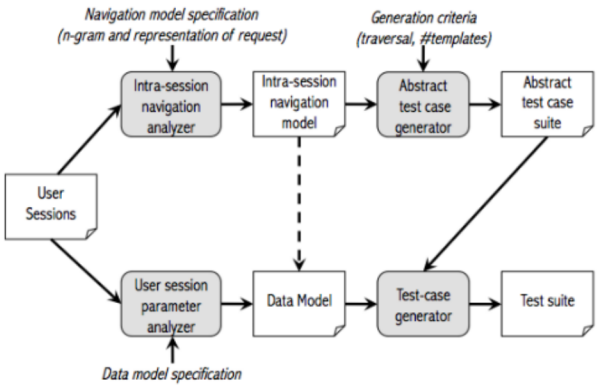


Figure 1. The Test Case Generation Process

over a network using HTTP. When a user's browser transmits an HTTP request to a web application, the application produces an appropriate response, typically an HTML document that the browser displays. The response can be either static, in which case the content is the same for all users, or dynamic such that its content may depend on user input or application state.

Before the test-case generation process shown in Figure 1 begins, the user requests are parsed and segmented to create user sessions. Each *user session* is a sequence of user requests in the form of base requests and name-value pairs. The request recorder treats hidden parameters the same as regular parameters. When cookies are available, we use cookies to generate user sessions. Otherwise, we say a user session begins when a request from a new Internet Protocol (IP) address arrives at the server and ends when the user leaves the web site or the session times out. We consider a 45 minute gap between two requests from a user to be equivalent to a session timing out [18].

From a set of user sessions and a navigation model specification, the *intra-session navigation analyzer* constructs an intra-session navigation model. The *abstract test case analyzer* uses the navigation model and abstract test case criteria to produce a set of abstract test cases. The abstract test cases are input into the *test case generator* with the data model to output a set of test cases—the test suite. The data model is constructed through an analysis of the user sessions. Various data models can be used to generate parameter values [17]. A tester can generate many test cases from one abstract test case by adding parameter values generated from different data models (or even the same model, if it is nondeterministic) to the test cases.

2.2. Usage-based Navigation Models

Several researchers have proposed representing web applications via navigation models. Wang et al. essentially

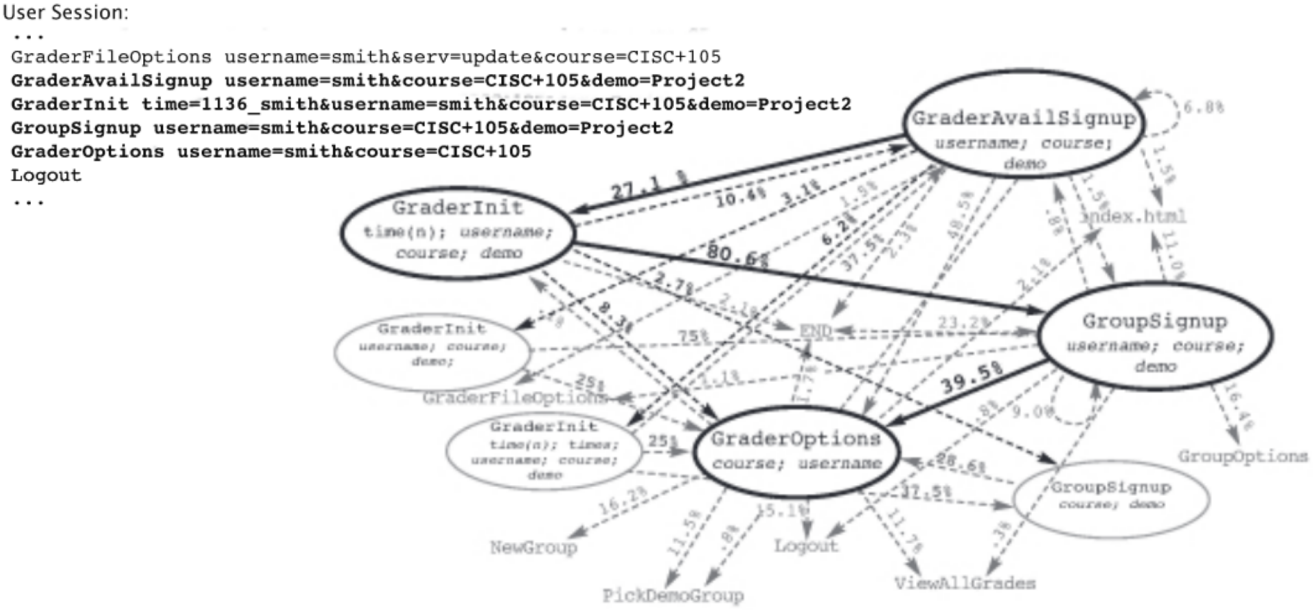


Figure 2. 2-gram Navigation Model for a Set of User Sessions

spider the application from a defined start page and use a combinatorial approach to input values into the application's forms to generate the navigation model [20]. Their model does not leverage usage information. Tonella and Ricca's navigation is similarly generated by spidering the application from a start page and inputting values from equivalence classes into the forms [19]. They augmented their navigation model with usage information, adding usage-based probabilities to the edges. Neither Wang et al.'s nor Tonella and Ricca's navigation model generation is completely automated. Both require the values to be input into forms to be known beforehand. Another limitation of both approaches is that neither seems to explicitly handle navigation that may depend on different application state (e.g., if a search fails to find any matches because of the contents of the database).

In this paper's study, we base our navigation models on the Sant et al. approach to generating test cases from a statistical model of user sessions [17]. The *navigation model* (called the "control model" in Sant et al. [17]) is an n-gram Markov model of the user sessions' requests, where n is one more than the number of previous requests used to predict the next request. A navigation model where n=1 (called 1-gram or *unigram*) models the probability of a user making a request, regardless of previous requests. For $n \geq 2$, the navigation model is a directed graph, where the states are the set of n-1-length request sequences in a set of user sessions. A state represents a particular request sequence and its outgoing edges are labeled with the conditional probabilities of the next request. The generated model may have several "start" states, since a user's session may start from any page, e.g., a bookmarked page, a page found by a search engine, or from continuing a timed-out session. In our preliminary studies, users entered an application using as many as 25 different pages.

A possible drawback of this approach is that the model is not complete unless users access all parts of the application. However, the goal of this kind of testing is not completeness but to focus on actual usage.

Figure 2 depicts part of a 2-gram (or *bigram*) navigation model derived from a set of collected user sessions to a deployed application, highlighting the requests from the user session in the upper left. The states represent the set of (length 1) user requests. For simplicity, we do not include the request type in the user session or in the states' labels in Figure 2. The edges represent transitions between sequential requests and are labeled with the probability of a user making the second request, given that they made the first request. For example, given that a user has made a request for `GroupSignup username;course;demo`, a user has a 39.5% chance of requesting `GroupOptions course; username` and a 16.4% chance of requesting `GroupOptions`.

2.3. Open Questions

2.3.1 Factors in Constructing Navigation Models

Figure 2 represents one way to create the navigation model. There are several key factors involved in building a usage-

based navigation model and generating abstract test cases from the model. The factors raise open questions, which we enumerate and then provide more background on the first three questions. In Section 3, we describe how we explored these important questions.

(a) How should user requests be represented in the model?
(b) How much history (i.e., what value of n) should be used in the n-gram Markov model of user session requests to predict the next request?
(c) How should the transitions between user requests be approximated in the model, i.e., how should the edges in the navigation model be labeled?
(d) How many user sessions should be used to build the model, and how does the model change as more user sessions are used?
(e) How does the model size vary with different configurations for building the model?

Representing user requests. An HTTP request consists of its *request type* (typically `GET` or `POST`), a *resource* (the 'R' in 'URL'), and optional parameter name-value pairs. For example, in the request `GET GraderOptions?user=smith&course=CISC101`, `GET` is the *request type*, `GraderOptions` is the *resource*, `user` and `course` are the *parameter names* and `smith` and `CISC101` are the *parameter values* for the parameter names `user` and `course`, respectively.

Sant et al. [17] represent a request by its **r**equest type, **r**esource, and parameter **n**ames of the data, which we refer to as RRN. Wang et al. [20] use a similar representation but do not explicitly include the request type.

Sant et al. [17] and Wang et al. [20] argue that representing requests by the resource and parameter names balances two concerns: (1) the ability to represent user navigation accurately and (2) maintaining the model's scalability as the number of user sessions and/or the size of the application increases. If we instead chose to represent requests by the resource alone, the size of the navigation model decreases (e.g., there would only be 4 states instead of 7 in Figure 2), but we would then need to design additional models of the request type and the parameter names to generate executable test cases. Another option is to include parameter *values*, in which case the model may become too large and restricts the generated test cases to the data in the collected user sessions; a tester may not want to be restricted in that way.

Amount of History. In an n-gram model, n is one more than the number of previous requests used to predict the next request. Intuitively, one would expect that as n increases, the navigation model would better represent users' navigation. However, as n increases, the size of the model will also increase because the possibilities for $n-1$ sequences increases. The tradeoffs of n have not been explored. From Sant et al.'s experiments [17], it is not clear if there is a benefit to using n greater than 1, with respect to code coverage, which is a surprising result. However, their experiments were limited to one small application and only looked at $n \leq 3$, and they did not analyze the effects of the navigation model and the data model separately.

Representing Inter-Request Navigation. The *abstract test case generator* generates abstract test cases by taking a weighted random walk of the navigation model according to the probability distribution learned from the user sessions and adding each visited state's last request to the abstract test case. Tonella and Ricca [19] suggest generating all possible paths through the model and ordering the paths by their probabilities. In general, we found our models were too large and complex (e.g., cycles) to generate all paths. An alternative using the weighted random walk is to consider the edges as all having equal probabilities, instead of using the usage-based probabilities.

2.3.2 Generated Abstract Test Cases

Different configurations for building the usage-based navigation model will most likely affect the characteristics of the abstract test cases generated from the model. Specifically, we note two open questions of concern to testers:

(a) How representative are the abstract test cases of the original user sessions?
(b) Do the abstract test cases enable testing of usage not found in the user sessions?

To evaluate how well the generated abstract test cases represent user sessions, we focus on the abstract test cases' coverage of sequences of requests from the original user sessions, which represents the users' navigation behaviors. A resource often maps to some application code. Regardless of the data model applied, the set of test cases that can be generated is dependent on the abstract test cases. If no abstract test case is generated with a specific sequence of resources (regardless of their representation), then *no test cases with that sequence will be generated* and, thus, will not cover any code associated with that sequence of resources. Therefore, we investigate the sequence coverage, of various lengths, provided by the abstract test cases generated by different navigation models.

3. Empirical Study

We designed an empirical study to explore some of the questions surrounding the factors in building usage-based navigation models and the effects on the resultant abstract

Subject	# of Classes	NCLOC
Masplas	9	609
Book	11	5279
CPM	76	7430
Logic	109	10718
DSpace	294	30847

Table 1. Subject Application Characteristics

test cases. Specifically, our empirical study seeks to answer the following questions:

1. **What are the most practical approaches to constructing usage-based navigation models for test case generation?** (Questions a-e in Section 2.3.1.)
2. **What are the characteristics of the generated abstract test cases for different model configurations and how can they be tuned for different testing goals?** (Questions a and b in Section 2.3.2.)

3.1. Subjects

In this paper, we target web applications written in Java using servlets and JSPs. The applications consist of a back-end data store, a Web server, and a client browser. Since our user-session-based testing techniques are language-independent—requiring user sessions but not source code for testing, our techniques can be easily extended to other web technologies.

We created 9 subject user-session sets from user requests to 5 publicly deployed applications. The applications were of varying sizes (1K-50K non-commented lines of code), technologies, and representative web application activities and usages: a conference website (Masplas); an e-commerce bookstore (Book) [9]; a course project manager (CPM); an online symbolic logic tutorial (Logic); and a customized digital library (DSpace) [7]. Book is the same application used in Sant et al.'s evaluation [17]. Table 1 summarizes the applications' code characteristics.

Book was the only application for which an email was sent to local newsgroups asking for volunteer users. These user requests were also used by Sant et al. [17]. We collected accesses for each application over a long period of time: Masplas: 2 months, CPM: 5 academic semesters, Logic: 1 academic semester, DSpace: 8 months.

We converted the user accesses into user sessions using Sprenkle et al.'s framework [18]. For CPM, we partitioned the user sessions by the semester in which they were collected to provide more test suite subjects to model and compare. Table 2 shows the characteristics of the collected user sessions, in terms of the number of user sessions (totalling over 3500 sessions), the number of user requests (totalling nearly 55K), and the percent of application code covered by the user sessions using Cobertura [5]. We report line coverage to show that the user sessions cover a large portion of the application.

Subject	# User Sessions	# Requests	% Lines Cvd
Masplas	169	1107	89%
Book	125	3564	61%
CPM1	58	1326	47%
CPM2	203	2393	66%
CPM3	105	1528	50%
CPM4	168	2240	58%
CPM5	356	4865	54%
Logic	517	16125	82%
DSpace	1800	22129	73%
Total	3501	55277	–

Table 2. Characteristics of User Session Sets

3.2. Navigation Model Analysis

In this section, we investigate the open questions in constructing navigation models of web applications. We discuss the results from studying how to represent user requests and inter-request navigation and then factors affecting model size. We generate the variations of navigation models for investigation by running the *intra-session navigation analyzer* on each set of user sessions. In the worst case, (DSpace, $n = 10$), the scripts generated the model in 7 minutes.

3.2.1 Methodology

We generated navigation models using 3 different representations of user requests: requestType+resource (RR), requestType+resource+parameter names (RRN), and requestType+resource+parameter names+parameter values (RRNV). We investigated the model characteristics for n from 1 to 10.

For each n, we measured the size of the navigation models, in terms of their number of states and edges, and computed statistics about the probabilities labeling the edges. We also generated navigation models incrementally to analyze how much the navigation model's size changes, to help determine when a tester could stop collecting user sessions. As user sessions are added to the navigation model, the size of the model may grow if the new user sessions contain requests to resources not accessed in the previous requests. We added user sessions in order of decreasing number of requests, which is an estimate of the best case because larger user sessions are more likely to contain requests that have not been seen before. Note that the order does **not** affect the final resulting model.

3.2.2 Results

Representing User Requests. Table 3 shows the number of states and edges for n=2, when representing user

	Repre-sentation	2-RR		2-RRN		Growth		2-RRNV		Growth over 2-RRN	
		States	Edges	States	Edges	States	Edges	States	Edges	States	Edges
Subjects	Masplas	23	218	36	264	56.5%	21.1%	191	489	431%	85.2%
	Book	10	71	27	173	170%	144%	910	1778	3270%	928%
	CPM1	49	185	53	200	8.16%	8.11%	471	860	789%	330%
	CPM2	61	254	98	333	60.7%	31.1%	609	1155	521%	247%
	CPM3	52	184	85	245	63.5%	33.2%	376	723	342%	195%
	CPM4	51	200	80	263	56.9%	31.5%	502	1018	528%	287%
	CPM5	53	220	72	265	35.9%	20.5%	893	1777	1140%	571%
	Logic	69	469	83	535	20.3%	14.07%	3716	7236	4380%	1250%
	DSpace	72	528	715	2172	893%	311%	6151	11200	760%	416%

Table 3. Comparing Request Representation: RR vs. RRN vs. RRNV

requests using the requestType+resource (RR), requestType+resource+*parameter names* (RRN), and requestType+resource+parameter names+*values* (RRNV) and the percent increase in both states and edges when adding information (parameter names and values, respectively) to the representation. Model growth percentages are to 3 significant figures.

The percentage increase when adding parameter names to the representation is less than 100% for most sets of user sessions. The growth is larger for DSpace because DSpace encodes some parameter names with values (e.g., `item_1234`), which results in many parameter names, e.g., as compared to just using `item`. The percentage increase when including parameter values is much higher, all above 100%. In general, as the number of user sessions increases, so will the percentage increase, as there will typically be a greater number of user-supplied unique values. We performed this analysis for $3 \leq n \leq 10$ but do not show the results due to space constraints. We observed that as the value of n increases, the percentage growth for both representations decreases, for all applications.

Our results provide empirical evidence for researchers' beliefs that representing user accesses by the resource and parameter names balances concerns of representation and scalability [17, 20]. Using RRN, there is typically less than a 100% increase in model size over RR and no need to develop a model of the parameter names, with its possible inaccuracies; however, there is still a need for a parameter value model to make the test cases executable. However, as noted above, the percentage increase in model size when including parameter values decreases as n increases. For the rest of this paper, we focus on models using RRN as the request representation.

Implication for testing: *When generating test cases automatically through a statistical model-based approach, separate the model of user behavior into two models and the test case generation process into two steps: (1) build a usage-based navigation model that represents the user accesses by resource and parameter name and is used to generate abstract test cases and (2) develop a separate data model for adding parameter values and use the data model to generate executable test cases. However, if the tester wants to use a sufficiently large n (application-dependent), using RRNV may not require too many additional resources than using RRN in terms of the model's size and does not require a data model.*

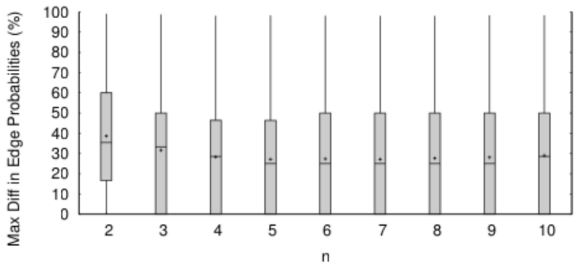


Figure 3. Distribution of Maximum Difference in Edge Probabilities Across All Applications

Representing Inter-Request Navigation. To answer the question about how transitions between user requests should be approximated in the model, we analyzed the range in the probabilities computed from usage information for each node's out edges, and compared with equal probabilities on its outgoing edges. For all nodes with at least two outgoing edges, we identified the edges with the minimum and maximum probabilities, respectively, and took the difference between those probabilities. The results of the edge analysis for all applications are in Figure 3. The x-axis represents the value of n used to generate the navigation model. The shaded box represents the middle 50% of the data, with whiskers extending to the 25th and 75th percentiles. The horizontal line denotes the median and $+$ represents the mean.

For half of the nodes (the median), the most likely edge is favored at least 25% more than the least likely edge (between 25.0% and 35.4%). Furthermore, for 25% of the nodes (the upper horizontal line in the shaded box), the most likely edge is favored at least 46% more than the least likely edge. The averages ranged between 27.1% and 38.6%. The

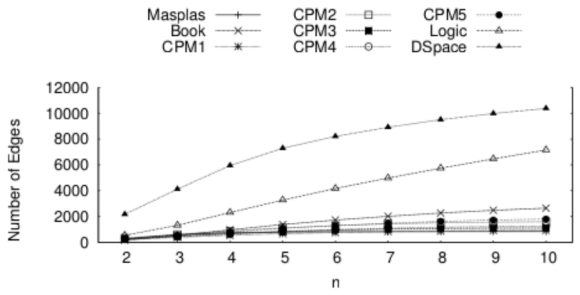


Figure 4. Number of Edges in Navigation Model as n increases

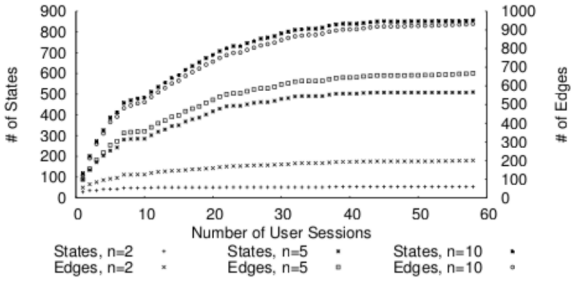


Figure 5. Navigation Model Growth for CPM1

maximum difference is larger for n=2. We did not observe any other distinct trends as n increases.

These results suggest that actual web application usage is considerably different from estimating user behavior without gathering usage information, such as by using equal probabilities on transitions.

Implication for testing: *While generating abstract test cases using a navigation model built without usage-based transition labels avoids the potential challenges of user log capture and analysis (e.g., privacy concerns or data collection effort), the resulting navigation model and abstract test cases will not align well with actual user behavior. Thus, if the tester's goal is to execute tests closely representing user behavior or prioritizing tests based on user behavior, the tester should gather usage logs and build a navigation model using those logs.*

Factors Affecting Model Size: n and the number of user sessions. Figure 4 shows how the navigation model's size grows as n increases in terms of the number of edges, for each set of user sessions. The x-axis represents the n used to generate the model, and the y-axis represents the number of edges in the resulting navigation model. As expected, as n increases, the number of states and edges increases similarly, thus we only show the graph for edges. We graphed the change in the slope of the graph as n increases and confirmed that the slope is decreasing for all sets of user sessions and, thus, does not have exponential growth. We expect that, as n increases beyond 10, the graph will eventually level out because the lengths of the user sessions and number of RRNs in an application are finite. These results suggest that the navigation models do not grow exponentially in size as n increases.

Implication for testing: *Model growth is not a good basis for determining which n to use for building the navigation model.*

As user sessions are added to the navigation model, the size of the model may grow if the new user sessions contain requests to resources not accessed in the previous requests. Figure 5 is representative of all sets of user sessions, in terms of states and edges, as more user sessions are used to build the navigation model, for n=2, 5, and 10. The x-axis represents the user session's id in decreasing size order. The left y-axis shows the number of states in the model, which represents all the user sessions up to and including this user session. The right y-axis shows the number of edges.

The slope of the graph when the user sessions are in decreasing length order reaches a point where it decreases. We observed this trend across all n and for all sets of user sessions. The result is not surprising because user sessions are known to be redundant [8, 16]. A tester could collect user sessions until the change in the model gets sufficiently small or until the user sessions meet some other criteria and generate the model. Furthermore, we were able to create stable navigation graphs from as few as 58 user sessions, as in CPM1, where stable means adding more user sessions does not add many new nodes or edges. However, as indicated by the model size in the figure, fewer sessions may result in stable models. These results indicate that building the navigation model from a relatively small number of user sessions and then processing more user sessions to augment the model only slightly increases the size of the resulting navigation model, likely due to high redundancy in users' navigation patterns.

Implication for testing: *A tester could stop gathering user sessions at any time after the model growth rate declines to a desired threshold and thus save time and space in collecting them. However, a tester may want to use as many user sessions as resources permit because additional user sessions provide more accurate estimates of users' transition probabilities, and the tester can throw out the user sessions and keep the smaller model.*

3.3. Abstract Test Case Analysis

This section presents our study of the generated abstract test cases from different variations of navigation models. We focus on how representative they are of the user sessions and how much they cover new navigations not in the

original user sessions. In the worst case, our abstract test case generator took 8 minutes to generate 500 non-duplicate abstract test cases.

3.3.1 Methodology

To account for the randomness of the random walk through the navigation model, the *abstract test case generator* generated a suite containing 500 non-duplicate abstract test cases for each navigation model. For the 1-gram model, we set the number of requests in a test case to 15 requests, based on the average sizes of the original user sessions. For $n > 1$, the number of requests in an abstract test case was determined by the weighted random walk of the navigation model. We generated a fixed number of abstract test cases instead of using a URL-coverage-based stopping criteria [15] so that we could compare results across applications and aggregate results without introducing bias towards a user session set.

For each abstract test case suite, we measured the number of the user sessions' RRN sequences of various lengths that are represented by the abstract test case suite, and the number of new, unique RRN sequences of various lengths that the abstract test cases explore.

3.3.2 Results

Representation of User Sessions. Figures 6(a) and 6(b) show the percentage of the user sessions' RRN sequences of various lengths that are represented in the abstract test case suites generated using n-gram models for n varying from 1 to 10, for two representative applications. The x-axis represents the length of the RRN sequence. The y-axis represents the percentage of the length-x RRN sequences in the user sessions that are represented in the abstract test cases. The lines show the data for the test cases generated from the various n-gram models.

In general, as the sequence length increases, the percent coverage decreases, for all sets of abstract test suites. This result is not surprising since a test suite can never cover more length-k sequences than length-k-1 sequences. Furthermore, as the n used to generate the model increases, the percentage of sequences covered by tests increases. We observe the same trend for all sets of user sessions. As n increases, the more history is used to predict the next request. Test cases generated from an n-gram model must be made up of sequences of length n-1 from the user sessions.

For all sets of abstract test suites for n=10 except for Logic and DSpace, the test suite covers 100% of RRN sequences of lengths 1 to 10. 100% coverage was surprising because we generated the abstract test cases using a weighted random walk and did not do anything to guarantee coverage. Logic and DSpace have larger (Figure 4), more complex navigation models, which we believe is one reason that the test cases for n=10 don't have 100% coverage. A tester could generate additional abstract test cases until reaching the desired coverage and reduce the abstract test cases or the tester could analyze the abstract test cases' coverage of the model and create test cases that traverse the uncovered sequences.

While not an explicit goal of our experiments, we were curious about the model coverage attained by the abstract test cases. We analyzed the abstract test suites' coverage of the navigation model's edges and found that coverage ranged from 43% for DSpace when n=3 to 96% for Book when n=9. Not surprisingly, we did not observe any trends as n increases because generation is random and based on probabilities.

In summary, in terms of the percentage of each user session's requestType+resource+parameter names sequences of lengths from 1 to 10 that are covered by the generated abstract test cases, the generated abstract test cases represent user navigation behavior very well in general.

Potential for Testing New Navigations. Figures 7(a) and 7(b) show the number of new, unique RRN sequences of various lengths that are represented in the generated abstract test cases that are not in the original user sessions, for two applications. The x-axis represents the length of the RRN sequence. The y-axis represents the number of length-x RRN sequences that are represented in the abstract test cases but not in the user sessions. The lines indicate the test cases generated from the n-gram models.

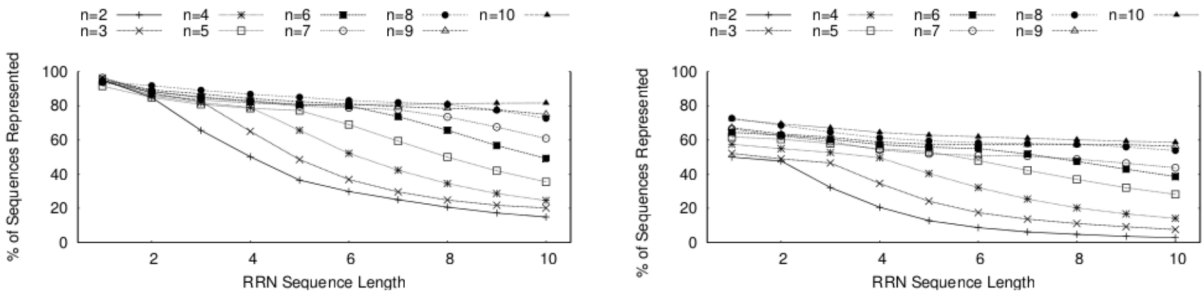
As expected, as n increases, the number of new sequences (of all lengths) decreases, with a few exceptions, for all sets of user sessions. The maximum number of new sequences is on the order of thousands for all sets of user sessions. Even with significant representation of user behavior in terms of sequences of all sizes, the generated abstract test cases also represent a significant number of new navigation paths of various lengths.

However, there is a tradeoff: as n increases, the representation of the actual user navigations increases, while the number of new sequences represented by the abstract test cases decreases (e.g., Figures 7(a)-7(b)).

Implication for testing: *If a tester wants to generate tests that are more closely aligned in navigation to the user sessions, then she should choose a larger n. If the tester is interested in more new navigations being tested, then she should choose a smaller n.*

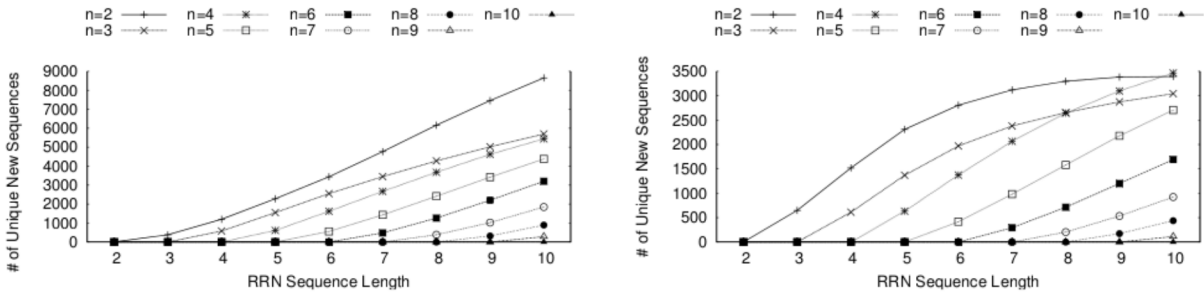
3.4. Threats to Validity

Since we performed our study on nine sets of user sessions for five applications, a study with additional sets of user sessions and applications may be necessary to generalize the results. However, we chose applications with differ-



(a) Logic (b) DSpace

Figure 6. % of RRN Sequences Represented in Abstract Test Cases



(a) Logic (b) DSpace

Figure 7. Number of New, Unique RRN Sequences in 500 Abstract Test Cases

ent usage characteristics, technologies, implementers, and application characteristics and gathered real user accesses to deployed web applications over a long period of time to improve the chances of seeing different usage patterns and to help reduce the threat to generalizing our results.

For the 1-gram control model, we chose for the abstract test cases to contain 15 requests. We could have selected a different number—for example, a larger number of requests would increase the chances of the abstract test cases' representing sequences from the original user sessions—or used an application-specific number of requests for each application. We chose a number that was comparable to the sizes of the user sessions across all applications.

4. Related Work

Earlier in the paper, we described the research most closely related to this work. The only other usage-based navigation model we are aware of is the usage-based navigation model that was developed for GUI testing [4]. A variation of this model may be applicable to web testing.

Approaches based on statically modeling web applications [1, 6, 13, 14] are challenged by the common dynamic features of web applications, particularly the dynamically generated pages and non-traditional control flow. Approaches based on modeling web applications with finite-state machines (FSMs) and using coverage criteria based on FSM test sequences are not meant to represent invalid inputs and suffer from the state explosion problem, which has been partially addressed by constraining inputs [2].

Halfond and Orso [10] mention that it would be interesting to combine their approach to testing web application interfaces with user-session-based testing. Their technique is based on static analysis of individual Java servlets and automatically discovers web application interfaces (i.e., sets of named input parameters with their domain type and relevant values, which can be processed as a group by a servlet) and then generates test cases by providing data values. This approach does not test sequences of requests or browser-based inputs but appears to be complementary to model-based test case generation.

Several groups propose applying concolic testing to web application testing to generate white-box-based test cases with the goal of achieving branch or bounded path coverage [3, 21]. Concrete and symbolic execution and constraint solving are combined to automatically and iteratively create new input values to explore additional control flow paths through a PHP script. We believe our insights can be applied to concolic testing-based approaches as they can be viewed as building a partial control model through user simulation of the application's menus and buttons.

5. Summary and Guidance to Testers

As web applications continue to grow in use and complexity, automatic test case generation becomes more important. In this paper, we present empirical evidence supporting specific design decisions in constructing usage-based navigation models for automatic generation of abstract test cases, to be used as a basis for executable test cases. In addition to substantiating some intuitive trade-offs and researchers' beliefs, our results suggest how the abstract test case suite can be tailored toward representing actual user behavior or potential new navigations.

Specifically, our results showed the following: with only a small number of user sessions, a tester can construct a navigation model that generates abstract test cases representative of observed user behavior as well as new behavior. A tester can tune the amount of history used (e.g., n) based on testing and representation goals. In general, a tester should use resource+parameter names to represent user requests, which generally balances scalability and representation requirements, but may want to consider including parameter values in the representation if using a larger n. Exponential model growth with a large number of user sessions is not a concern because of the redundancy in user sessions.

Armed with knowledge of how to best generate abstract test cases, we are currently investigating potential data models for generating parameter values to be integrated with the abstract test cases to construct effective executable test cases. We also plan to evaluate the impact of the abstract test cases on data models and the effectiveness of the resulting executable test suites in fault detection and code coverage in publicly deployed web applications.

6. Acknowledgments

We thank Lloyd Greenwald of Sant et al. [17] for clarifying their representation of user requests and implementation. We thank Natallia Robinson, who developed the Logic application. Finally, we thank Emily Hill for her help in analyzing the edges' probabilities and feedback on the paper.

References

[1] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Modelling methods for web application verification and testing: state of the art. *Software Testing, Verification, and Reliability*, 19(4):265–296, 2009.

[2] A. A. Andrews, J. Offutt, C. Dyreson, C. J. Mallery, K. Jerath, and R. Alexander. Scalability issues with using FSMs to test web applications. *Information and Software Technology*, 2009.

[3] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Int'l Symp. on Software Testing and Analysis*, July 2008.

[4] P. A. Brooks and A. M. Memon. Automated GUI testing guided by usage profiles. In *International Conference on Automated Software Engineering (ASE)*, 2007.

[5] Cobertura. `http://cobertura.sourceforge.net/`, 2010.

[6] G. Di Lucca, A. Fasolino, F. Faralli, and U. Carlini. Testing web applications. In *International Conference on Software Maintenance*, 2002.

[7] DSpace Federation. `http://www.dspace.org/`, 2010.

[8] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user session data to support web application testing. *IEEE Trans. on Software Engineering*, 31(3), 2005.

[9] Open source web applications with source code. `http://www.gotocode.com`, 2003.

[10] W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Foundations of Software Engineering*, 2007.

[11] S. Halle, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Automated Software Engineering*, 2010.

[12] C. Kallepalli and J. Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, 2001.

[13] C.-H. Liu, K. D. C., P. Hsia, and C.-T. Hsu. Structural testing of web applications. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2000.

[14] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Int'l Conf. on Software Engineering (ICSE)*, 2001.

[15] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock. Web application testing with customized test requirements—an experimental comparison study. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2006.

[16] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. S. Greenwald. Applying concept analysis to user-session-based testing of web applications. *Transactions on Software Engineering*, 33(10):643–658, October 2007.

[17] J. Sant, A. Souter, and L. Greenwald. An exploration of statistical models of automated test case generation. In *International Workshop on Dynamic Analysis*, May 2005.

[18] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. A case study of automatically creating test suites from web application field data. In *Workshop on Testing, Analysis, and Verification of Web Services and Applications*, 2006.

[19] P. Tonella and F. Ricca. Statistical testing of web applications. *Journal of Software Maintenance and Evolution*, 16(1-2):103–127, 2004.

[20] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence. A combinatorial approach to building navigation graphs for dynamic web applications. In *International Conference on Software Maintenance*, 2009.

[21] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Int'l Symp. on Software Testing and Analysis*, 2008.